Distributed Location Databases for Tracking Highly Mobile Objects

Evaggelia Pitoura and Ioannis Fudos

Department of Computer Science, University of Ioannina, GR 45110 Ioannina, Greece
Email: {pitoura,fudos}@cs.uoi.gr

In current distributed systems, the notion of mobility is emerging in many forms and applications. Increasingly many users are not tied to a fixed access point but instead use wireless communications or dial-up services to access data independent of their physical location. Furthermore, mobile software, a popular example being mobile agents, is frequently used as a new form of building distributed network-centric applications. Tracking mobile objects, i.e. identifying their current location, is central to such settings. In this paper, we exploit the use of hierarchical distributed location databases, where each database site covers a specific geographical region and contains location information about all objects residing in it. For highly mobile objects, a scheme based on forwarding pointers enhanced with auxiliary caching techniques is presented, to reduce the cost of the overall network and database traffic generated by frequent location updates. The scheme is extended to support concurrency and failure recovery. Performance is demonstrated through a number of simulation experiments for a range of call to mobility ratios and for a variety of moving and calling behaviors.

1. INTRODUCTION

Advances in wireless telecommunications and in the development of portable computers have provided for the emergence of wireless mobile computing [1, 2, 3]. In wireless mobile computing, hosts are not attached to a fixed geographical location, instead their point of attachment to the network changes as they relocate from one support environment to another. Besides mobility tied to wireless communications, mobile software (code or data that move among network sites) is emerging as a new form of building distributed network-centric applications. The mobile software agent [4, 5] is a popular form of such mobile software. Mobile agents are processes that may be dispatched from a source computer and transported to remote servers for execution. Mobile agents can be launched into an unstructured network and roam around to accomplish their task [6], thus providing an efficient, asynchronous method for collecting information or attaining services in rapidly evolving networks. Other applications of moving software include the relocation of a user's personal environment to support ubiquitous computing [7], and the migration of services to support load balancing, for instance the active transfer of web pages to replication servers in the proximity of clients [8].

Since delivering any message to mobile objects requires locating them first, deriving efficient strategies for tracking moving objects is central to mobile computing. In particular, for future personal communication service (PCS) systems, with high user populations and numerous customer services, such signaling and database traffic for locating users is expected to increase dramatically [9]. To maintain the current location of mobile objects, distributed location databases or directories are deployed. Such databases are often hierarchically structured [9, 10, 11, 12] to accommodate the increased traffic associated with locating moving objects. In particular, each leaf database covers a specific geographical region and maintains location information for all objects currently residing in that region. Location databases at internal nodes of the hierarchy contain information about the location of all objects registered in areas covered by the databases at their children nodes.

In this paper, we consider the problem of locating moving objects using such hierarchical tree-structured location databases. In particular, to reduce the cost associated with location updates, instead of updating all location databases involved, only databases up to a specific level of the tree are updated and a forwarding pointer is appropriately set at some lower level database. However, if forwarding pointers are never deleted, then long chains are created, whose traversal results in an excessive increase in the cost of locating objects. We propose and evaluate two alternative conditions for purging forwarding pointers and for updating the hierarchical database: one based on the maximum number of the forwarding pointers and the other on the physical distance between the sites in the chain. In addition, we introduce a number of auxiliary caching techniques that effectively reduce the number of pointers traversed before locating a moving object.

We extend the scheme to correctly support concurrent execution of location tracking and updating. The treatment of concurrency is general enough and is applicable to any

hierarchical location database. We also discuss recovery in the case of site failures and propose solutions to this end.

To study the performance of the forwarding scheme in conjunction with the caching techniques and the various conditions for purging forwarding chains, we have developed an event-driven simulator. We consider two basic operations: calls to mobile objects that initiate location searches and moves of mobile objects that initiate location updates. We have performed a number of experiments for a range of call to mobility ratios and for objects with varying mobility and calling behavior. The cost of each operation has two components: a cost associated with database operations and a cost associated with messages among sites. Since which of the two components is the dominating factor follows from various system-dependent parameters, our analysis treats each of these costs separately. The results clearly show that under certain assumptions and for small call to mobility ratios, the forwarding scheme coupled with appropriate auxiliary caching techniques on a per object basis can reduce both the overall database load and the communication traffic by 20% to 60% depending on the call to mobility ratio.

The rest of this paper is organized as follows. In Section 2 we introduce the forwarding scheme and various strategies for improving its performance. In Section 3 we extend the proposed protocol to handle correctness issues that arise from the concurrent execution of location searches and updates. In Section 4 we discuss replication and advanced location queries. In Section 5 we present our models for the call and mobility behavior, briefly describe our cost model, and report performance results. In Section 6 we compare our work with related research and in Section 7 we offer conclusions.

2. THE LOCATION STRATEGY

To support efficient tracking of mobile objects, their current location is maintained in a distributed database. Two operations are central: updating the stored location of an object when the object moves and searching the location of an object when the object is called or invoked. The granularity of the information maintained about the location of an object varies. It can be a single site or a set of sites. In the latter case, once the set is identified, the object is located by broadcasting a message to all sites in the set, an operation called paging. We call such sets of sites *logical cells* or l-cells.

In particular, in cellular architectures, mobile users are located in system-defined cells, which are bounded geographical areas. A cell is a uniquely identifiable unit. The location strategies proposed in the IS-41 and GSM [13] standards use a two-tier system in which each moving user is associated with a pair of a home location register (HLR) and a visitor location register (VLR). Calls to a given user query first the VLR in the caller's region, and then, if the callee is not found in this region, the callee's HLR is queried. Moves require updating the HLR. Similar approaches are readily applicable to software, for example mobile agents.

In this case, the stored location is a network site or the agent's current context. Location information may be stored at the agent's home, for example at its birth site, as well as at its visiting site, i.e. the site it is currently in. In cellular architectures, an l-cell covers a number of neighbor cells, while in other architectures an l-cell may cover few interconnected LANS.

2.1. Hierarchical location databases

To avoid 'long-distance' signaling messages to the HLR, when most location searches and updates are geographically localized, a hierarchical directory structure has been proposed [9, 10, 11] and is under study for the European third-generation mobile system called the universal mobile telecommunication system (UMTS) [14]. Similar hierarchical architectures have been proposed for locating moving objects in wireline distributed computing as well (for example, [12]).

Hierarchical location schemes extend two-tier schemes by maintaining a hierarchy of location databases, where a location database at a higher level contains location information for objects located at levels below it. A location database at a leaf serves a single l-cell and contains entries for all objects currently in this l-cell. A database at an internal node maintains information about objects residing in the set of l-cells in its subtree. For each mobile object, this information is a pointer to an entry at a lower level database. For example, in Figure 1, for an object x residing at l-cell 18, there is an entry in the database at node 0 pointing to the entry for x in the database at node 2. The entry for x in the database at node 2 points to the entry for x in the database at node 6, which in turn points to the entry for x in the database at node 18.

It is often the case that the only way that two nodes can communicate with each other is through the hierarchy; no other physical connection exists among them. For instance, in telephony, the databases may be placed at the telephone switches.

Such hierarchical schemes lead to reductions in communication cost when most calls and moves are localized. In such cases, instead of contacting the HLR of the object that may be located far away from the object's current location, a small number of location databases in the object's neighborhood are accessed. In addition, there is no need for binding an object to a permanent HLR, since the object can be located by querying the databases in the hierarchy. On the other hand, the number of database operations caused by call and move operations increase. Table 1 summarizes some of the pros and cons of hierarchical architectures. One problem with hierarchical architectures is that the load is not evenly distributed among all nodes since more load is imposed on nodes on higher levels. Approaches that distribute the entries of higher-level nodes at more than one site are applicable but are not discussed here (see [15]).

Let $LCA(i, j)$ stand for the least common ancestor of nodes i and j, and $lca(i, j)$ be the level of the $LCA(i, j)$. Let the level of the leaves be level 0. When object x moves
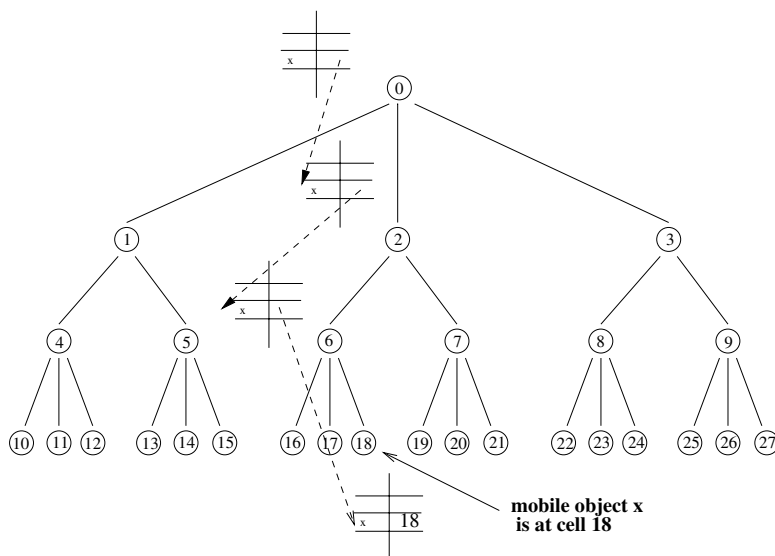
FIGURE 1. Hierarchical location schema. Location database entries are pointers to lower level databases.

TABLE 1. Summary of the pros and cons of hierarchical architectures.

(+)	No need for life-long numbering (no pre-assigned HLR)
(+)	Support for locality
(−)	Increased number of operations (database operations and communication messages)
(−)	Increased load and storage requirements at higher levels

from l-cell i to l-cell j, the entries for x in the databases along the path from j to $LCA(i, j)$, and from $LCA(i, j)$ to i must be updated. For instance, when object x moves from 18 to 20, the entries at nodes 20, 7, 2, 6 and 18 are updated. Specifically, the entry for x is deleted from the databases at nodes 18 and 6, and an entry for x is added to the databases at nodes 2, 7 and 20. When a caller located at l-cell i places a call for an object y located at l-cell j, the lookup procedure queries databases starting from node i and proceeding up the tree until the first entry for x is encountered. This happens at node $LCA(i, j)$ (the least common ancestor of nodes i and j). Then, the lookup procedure proceeds downwards following the pointers to node j. For instance, a call placed from l-cell 21 to object x (Figure 1), queries databases at nodes 21, 7 and finds the first entry for x at node 2. Then, it follows the pointers to nodes 6 and 18. We call the above operations *basic move* and *basic call* respectively.

2.2. Forwarding pointers

To reduce the update cost, a forwarding pointer strategy is deployed. With this approach, instead of updating all databases on the path from j through $LCA(j, i)$ to i, only the databases up to a level m are updated. In addition, a pointer is set from node s to node t, where s is the ancestor of i at level m, and t is the ancestor of j at level m. The level of s and t may vary. A subsequent caller reaches x

through a combination of database lookups and forwarding pointer traversals. Take, for example, object x located at node 14 that moves to node 17 (Figure 2). Let level $m = 2$. A new entry for x is created in the databases at nodes 17, 6 and 2, the entries for x in the databases at nodes 14 and 5 are deleted, and a pointer is set at x's entry in the database at node 1 pointing to the entry of x in the database at node 2. The entry for x at node 0 is not updated. Using forwarding pointers may increase the cost of calls, since it may result in a combination of tree links and forwarding pointers traversals. For instance, when an object, say at l-cell 22, calls x, the search message traverses the tree from node 12 up to the root node 0 where the first entry for x is found, then goes down to 1, follows the forwarding pointer to 2, and traverses downwards the path from 2 to 17. On the other hand, a call placed by an object at 15, results in a shorter route: it goes up to 5 and 1 and then to 2, 6, and 17.

In the following, we assume forwarding at the leaf level in which case forwarding pointers are set among leaf nodes, but the analysis applies to other types of forwarding as well. In this case, when an object moves from location i to location j, a forwarding pointer is simply set at node i pointing to node j. Calls originating from a l-cell k to an object x located at l-cell l proceed initially as in the basic call: go up the tree till the first entry for x is encountered and then down following the entries to a leaf node, say m. However, x may not be actually located at m; instead the entry at m may be a forwarding pointer to another leaf node. Thus, before actually locating an object, a chain of forwarding pointers may need to be traversed.

2.3. Caching

We propose auxiliary techniques that reduce the number of forwarding pointers that a call traverses before locating a mobile object. The first technique is based on the observation that it should be possible to re-use the
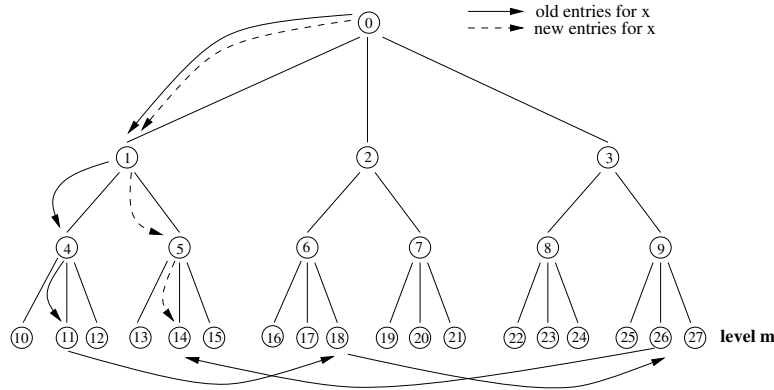
FIGURE 2. Pointer forwarding.

information about the object's location from the previous call to that object. To this end, each time a call is set up, after the address of the callee has been resolved, it is saved at the first node of the chain. Thus, any subsequent call to that object can use the cached location to locate the object, instead of traversing a number of forwarding pointers. This is essentially a variation of caching in which, instead of caching the address of the callee at the caller [16], the address is cached at the starting node of the chain of forwarding pointers. The cached location can then be used not only by subsequent calls from the same l-cell, but also from calls originated from any l-cell. Caching does not add to the latency of a call operation, since it can be performed off-line. In contrast to caching at the caller's location, in which case cache entries become obsolete once the callee moves, the cached entry at the first node of the chain can still be used in conjunction with the forwarding pointers even when the callee moves.

A similar caching strategy can be deployed along with move operations. In particular, during each move, the new location of the object can be cached at the first node of the chain. However, in contrast to caching at calls, the entry cached at a move is used only if the next operation involving the object is a call operation. Intuitively, if the call-to-mobility ratio (*CMR*) is very low, continuously updating the cached entry at each and every move operation (which corresponds to having in effect a chain of one pointer) adds to the overall traffic without necessarily reducing the latency of some call operation. Thus, instead of caching at each move, caching is done selectively.

The condition for caching at a move is based on an estimation of the probability that the next operation will be a call operation. This is approximated by comparing the number of consequent moves and the call-to-mobility ratio. This requires a counter to be associated with each mobile unit. The counter is incremented when the unit changes location and is set to zero when the unit receives a call. It also assumes that an estimation of the *CMR* is available. For this purpose, techniques such as those proposed in [16] can be utilized. Let *number_of_moves* be the value of the

counter and *estimated_cmr* be the value of the current *CMR*, then a simple heuristic is to cache when *number_of_moves* ≥ *estimated_cmr*. Again, caching can be performed off-line without adding up to the move set-up time.

2.4. Full updates

Besides caching, in order for the hierarchical directory scheme to maintain the property of efficiently supporting local operations, the directory entries must be occasionally fully updated. Otherwise, the first node of the chain, i.e. the node at which each call is initially directed by following the tree database entries, may end up being far away from the object's actual location. Fully updating the data structure includes (a) deleting all intermediate forwarding pointers in the chain and (b) updating the internal nodes of the tree (Figure 3).

Regarding full updates, two issues must be addressed. One issue is the correct execution of calls that proceed concurrently with a full update. We consider this issue in Section 3.1. The other issue refers to the condition for initiating these procedures. We consider two possibilities. The first is to initiate a full update when the number of forwarding pointers exceeds some maximum value. This is reasonable since the size of the chain directly affects the cost of calls because this cost depends on the number of pointers to be traversed. The other is to fully update when the object moves outside the vicinity of its current region, i.e. when locality deteriorates. We capture this by considering the level of the least common ancestor of the previous and the new location. In particular, an update is initiated when the object moves to a location *u* such that the level of the least common ancestor of *u* and the previous location is larger than a threshold level. We experimented with different values for both criteria for updating; the results are presented in Section 5.3.

Before setting a forwarding pointer, detecting cycles is necessary to avoid infinite loops during calls. For example, consider chain 11 → 18 → 26 → 14 and a move made to 18. Carelessly adding a pointer to 18 results in chain
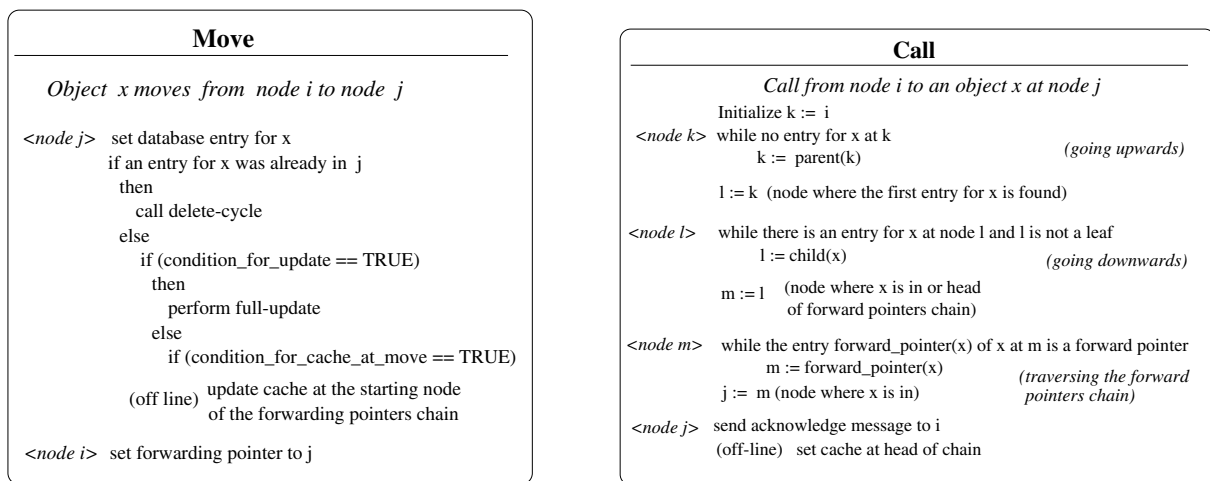
FIGURE 3. Full update of the directory entries.

FIGURE 4. Move and call procedures. For clarity of presentation, for the call operation we do not show the case in which node i is one of the nodes in the forwarding pointers chain.

$11 \rightarrow 18 \rightarrow 26 \rightarrow 14 \rightarrow 18$. Cycles can be detected by checking upon each move of an object A from j to i, whether an entry for A already exists at i. If so, there is a path from j to i which can then be purged, and the chain of the example becomes $11 \rightarrow 18$.

Finally, although during normal moves cache entries never become invalid, moves that form cycles or cause the initiation of full updates make cached entries obsolete. Thus, there is an additional overhead associated with such moves, that of updating the entry cached at the first node of the chain, if any.

Figure 4 summarizes the move and call procedures. The deployment of these techniques along with the forwarding pointers strategy reduces the total database load and communication traffic over the basic hierarchical scheme by a considerable factor (see Section 5).

3. CONCURRENCY CONTROL AND RECOVERY

3.1. Correctness of concurrent operation

In a hierarchical location database, each call results in a number of query operations being issued at various nodes of the database. Similarly, a full update initiated by a move

causes update operations to be executed at several nodes of the database. The discussion so far was based on the assumption that moves and calls arrive sequentially and are handled atomically and isolated one at a time, and thus, there is no interleaving between the database queries and updates initiated by move and call operations. In this section, we extend the move and call procedures to handle concurrency.

First, when a move from l-cell i to l-cell j occurs, a forwarding pointer pointing to j is set at i. This forwarding pointer ensures that any calls that have been issued prior to the movement and have been directed to the old address i will not be lost. Then, *a specific order* is imposed on the database operations that are initiated by a full update. In particular, each full update is performed in two phases: an *add* and a *delete* phase. Let an object's x movement from i to j that causes a full update, and let v_0 be the first node in the forwarding pointers chain of x. If there is no such chain, v_0 is i. First, in the add phase, entries at the path from j to $LCA(j, v_0)$ are inserted in a bottom-up fashion. Then, in the delete phase, the entries at the path from $LCA(j, v_0)$ to v_0 are removed in a top-down fashion. In a similar manner, forwarding pointers from v_0 to i, if any, are deleted starting from v_0 and moving towards i. We also distinguish between

the two phases of a call. During the *upward phase*, a call moves up the tree to find an entry for the callee. Once the first entry for the location of the callee is found, the call moves *downwards* to this location.

We serialize full updates caused by move operations as follows: a full update operation P_k caused by a movement of an object x is initiated only after any database updates caused by any previous full update operation P_m of the location of x has been completed. To enforce the serialization of full updates, P_k locks for updates the database entry for x at the new address j. When an entry is locked for updates by an operation P_k, it cannot be updated by any other operation P_l $(P_l \neq P_k)$. Instead, any such update operation is blocked until the lock is released. A message is sent by P_k to unlock j, when the last entry for x is deleted. Only then, can a new full update operation P_l proceed.

Besides updates, we must also ensure that any call concurrent with a full update will succeed in locating the object. Specifically, we will first show that:

LEMMA 1. *During the upward phase, any call placed by an object at l-cell c and being concurrently executed with a move, say from location i to location j, will be directed either to the old location, i, or the new location j.*

Proof. To distinguish among the possible relative positions of c, i, and j, we use the following two properties of the least common ancestor that hold for any tree nodes x, y, and z: (a) $lca(x, y) = lca(x, z) \Rightarrow lca(y, z) \leq lca(x, z)$, and (b) $lca(x, y) > lca(x, z) \Rightarrow lca(y, z) = lca(x, y)$.

Case (i): $lca(j, i) < lca(i, c)$, that is, the caller is equidistant from the old, i, and the new, j, locations. Then, during its upward phase, the call encounters the first entry for x at $LCA(j, c)$ $(= LCA(i, c))$. Then, at node $LCA(j, i)$, the call moves towards j if the add phase has reached the node, and towards i otherwise.

Case (ii): $lca(j, i) > lca(j, c)$, that is, the caller is closer to the new location. The call proceeds upwards to node $LCA(j, c)$. If the add phase has reached $LCA(j, c)$, then the call is directed to j. Otherwise it continues moving upwards till it either reaches a newly added entry or it reaches $LCA(c, i)$, whichever comes first. In the former case, the caller is directed to j and in the latter case to i.

Case (iii): $lca(j, i) = lca(j, c)$, that is, the caller is closer to the old location. If the delete phase has reached node $LCA(i, c)$, the call moves up to $LCA(j, c)$ and then towards j. Otherwise it moves towards i. □

Now, we must show that the call during its downward phase will succeed in locating the callee. There is a racing issue. To illustrate it, assume the concurrent execution of a full update for object x being in its delete phase and a concurrent call to object x being in its downward phase. Say that at some time, the call queries node m at level k and follows the pointer to the corresponding level $k - 1$ node, say node l. Immediately afterwards, the move deletes the entry at level k. Then, traveling faster than the call, the move arrives at node l and deletes the entry for x. When the call

arrives at node l, it finds no entry for x. There are at least three ways to handle the problem. We adopt the third as being the most practical.

Method 1: Repeatable calls. In this approach, no special treatment is given to concurrent operations. Calls that read obsolete data fail to track the object, and the lookup procedure is reissued anew. Specifically, a search fails at some node u at level k, if the callee has moved out of the subtree rooted at u and the corresponding entries have been deleted. In this case, the call can continue by moving upwards to level $k + 1$ and read the database at the parent of node u. Unless, in the meanwhile, the callee has moved again in the subtree rooted at u, the call will succeed this second time. Otherwise, it will move repeatedly up and down the tree. Thus, although simple, this method does not provide any upper bound on the number of tries a call has to make before locating a moving object.

Method 2: Obtaining locks. We can use traditional lock-based concurrency control protocols. In this case, there are two types of locks: read and write locks. A read lock for an entry is granted if no other operation holds a write lock on the entry, while a write lock is granted if no other operation holds a read or write lock on the entry. The simplistic locking strategy would be for both the call and the full-update operations to request and acquire read and write locks, respectively, on all nodes involved before proceeding. We discuss here how a locking scheme could be used if we take into consideration the fact that full updates are serialized. If we assume the two phases of the full-update operation, in the beginning of the delete phase, a full-update operation must request and obtain write locks at *all* nodes on the path to the actual location of the object, before deleting any entry. After attaining all such locks, the update operation can release them. Before querying any node, a call operation must first request and obtain a read lock on the associated entry at the specific node. Read locks can be released immediately after the entry is read. Such a scheme may result in high lock contention in the higher levels of the hierarchy. Furthermore, in the case of updates, locks must be acquired in a number of nodes of the distributed architecture, before the update proceeds. A more efficient approach is to use counters.

Method 3: Using counters. We associate with each entry for x at each node m a counter, called pending calls counter (PCC_m). PCC_m counts the number of calls that queried the database entry for x at node m while moving downwards to locate x or following the chain of forwarding pointers towards x's current location. An additional counter at m, called $local_m$, counts the number of calls whose downward phase started at node m, i.e. calls that found the first entry for x at node m. Let n be the child node of m that contains an entry for x. Then, $PCC_m + local_m - PCC_n$ is equal to the number of call operations traveling downwards from node m to node n. For example, in Figure 5, the fact that $PCC_1 = local_0 + PCC_0$ indicates that there are no pending calls for x from node 0 to node 1, that is there are no call operations to
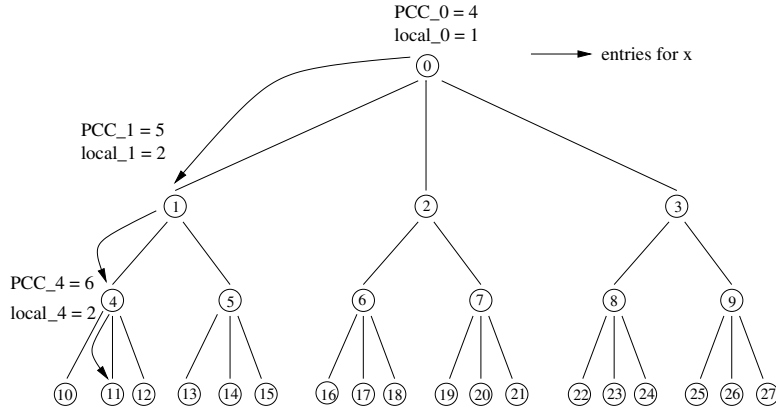
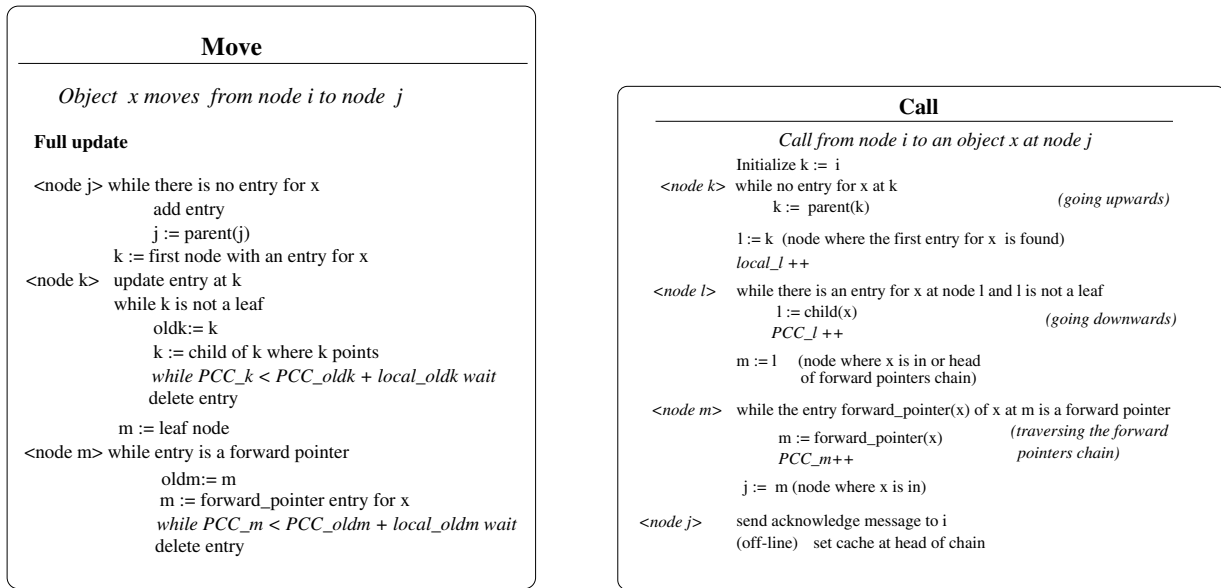**FIGURE 5.** PCC counts the number of calls traveling from a node to its child.

Move

Object x moves from node i to node j

Full update

```
<node j>  while there is no entry for x
                 add entry
                 j := parent(j)
          k := first node with an entry for x
<node k>  update entry at k
          while k is not a leaf
                 oldk:= k
                 k := child of k where k points
                 while PCC_k < PCC_oldk + local_oldk wait
                 delete entry
          m := leaf node
<node m>  while entry is a forward pointer
                 oldm:= m
                 m := forward_pointer entry for x
                 while PCC_m < PCC_oldm + local_oldm wait
                 delete entry
```

Call

Call from node i to an object x at node j

```
          Initialize k := i
<node k>  while no entry for x at k
                 k := parent(k)                    (going upwards)

          l := k  (node where the first entry for x is found)
          local_l ++
<node l>  while there is an entry for x at node l and l is not a leaf
                 l := child(x)
                 PCC_l ++                          (going downwards)

          m := l   (node where x is in or head
                    of forward pointers chain)
<node m>  while the entry forward_pointer(x) of x at m is a forward pointer
                 m := forward_pointer(x)           (traversing the forward
                 PCC_m++                            pointers chain)

          j := m (node where x is in)
<node j>  send acknowledge message to i
          (off-line)  set cache at head of chain
```

FIGURE 6. Move and call procedures extended for handling concurrency.

x that have queried node 0 but have not yet reached node 1. Correspondingly, the fact that $PCC_1 + local_1 - PCC_4 = 1$ indicates that there is one pending call from node 1 to node 4.

During the delete phase of a full update operation, an entry at a node n is deleted only if $PCC_n = PCC_m + local_m$, where m is n's parent node whose entry for x points to n. Otherwise, the delete operation is delayed until the calls searching for x arrive at node n. We assume that the operations on counters are atomic.

LEMMA 2. *During the downward phase, a call moves correctly to the location found in the upward phase. Thus, it arrives at j either directly or indirectly through i by following pointers.*

Proof. Let a call c be at node m at level k during its downward phase after having found an entry for x. If the entry for x at m directs the call to the new location j, then the call will succeed in finding an entry at level $k - 1$ since in the add phase the entries are included in a bottom-up fashion, and thus the entry at level $k - 1$ will be the updated one.

If the entry directs the call to the old location i, the call will succeed finding an entry at level $k - 1$ since an entry is deleted only when all pending calls from previous levels have been serviced as indicated by the PCC. Similar claims hold for following the forward chain, if any. □

Thus, all calls will eventually succeed in reaching either i or j. In effect, the algorithm in the first case makes a call appear as if it has occurred before the move, that is, it pushes the call backward, or makes it appear as if it has occurred after the move, that is, it pushes the call forward. Figure 6 summarizes the required modifications of the call procedure and the implementation of the full update operation.

The condition $PCC_n = PCC_m + local_m$ will eventually hold, if no messages are lost. Assume that x moves from cell i to cell j, and $LCA(i, j)$ is at level k. After the entry of x at node $LCA(i, j)$ is updated, no more messages will be directed to the entry for x at node n at level $k - 1$ (that is, neither PCC_n nor $local_n$ will be incremented further). We can similarly show that this holds for all levels k down

to the leaf level. That is after the entry at level $k + 1$ ($k \geq 0$) is deleted no more messages will be directed to the level k database, thus eventually PCC_n will be equal to $PCC_m + local_m$. However, even if messages are lost, this has no effect other than leaving a few obsolete entries; no call or move operation is blocked.

3.2. Failure recovery

With regard to communication failures, we assume that low level communication protocols ensure that (a) there are no errors in messages and (b) no messages are lost. Next, we consider site and media failures. In the case of site failures, one or more database sites become unavailable, however, location data stored on disk are not lost and become available upon recovery. In the case of media failures, location data are permanently lost. In both cases, the replication inherent in the hierarchical model is exploited for efficient recovery.

3.2.1. Internal node failure

When an internal database site becomes unavailable, any location query for an object x directed to this site is re-directed to all children of the site. The call procedure proceeds downwards from the child at which an entry for x is found. When a database site is unreachable, any requests to insert or delete entries at this site are logged at the sending site j. The updating process is suspended till the site recovers. Following recovery, the update process resumes from site j.

Upon recovery, information at an internal node can be reconstructed by combining information from its children. To this end, node i sends a request to *reconstruct* to its children. Their replies are unioned to reconstruct the location entries at i.

When two or more children contain values for the same object x then:

(i) when no actual position entry and no head of a forwarding pointer chain is contained, then i should have no entry for object x;

(ii) when an actual position entry is contained for object x, then i consults its parent. If i's parent contains an entry for object x pointing to i, then i creates an entry for x pointing to i, otherwise it ignores this child's reply;

(iii) otherwise (there is a head of a forwarding pointer chain and no actual position entry), the reply of the child that contains the head of a forwarding pointer chain for x is adopted.

There is one subtle point. Consider the following scenario. A delete message for object x is sent from node i to its child j, then i fails. Shortly after, i recovers and sends a reconstruct message to j that arrives before the previously sent delete message. In this case, the reply from j and thus the reconstructed location database at i includes an entry for x. Later, when the delete message arrives at j, the entry at j will be deleted. Thus, queries for x arriving at j from i will fail to find an entry for x. A simple solution is to augment delete and reconstruct messages with timestamps.

Each site j maintains the timestamp t_{rec} of the most recent reconstruct message received. If a delete message with timestamp smaller than t_{rec} arrives at j, j first sends a delete message to its parent. It deletes the entry only after the message is acknowledged. Thus there is no need to maintain backups of internal database sites, since even in the case of media failures, information from children database sites suffices to reconstruct the location database at the site.

3.2.2. Leaf node failure

In the case of a search arriving at a leaf database site that failed, first all sites in the l-cell covered by the failed site are paged. If the object is not found, then in the case of a search originated from a caller at this site, the search operation proceeds upwards to the parent of the site as in the normal case. If, however, the search arrives from a parent or sibling site, indicating that the search followed a tree or forward link respectively, a global search is necessary to locate the moving object, when paging fails. Some form of informed search, such as searching sites in the neighborhood, is due. Alternatively, the call may be suspended till the database site is reconstructed. When an object enters a new l-cell or leaves an l-cell, whose covering location database has failed, the object is informed to periodically poll the failed site to install the location updates. Upon recovery after a site failure, obsolete entries created by objects entering or leaving l-cells during the failure, are expected to be updated through such periodic polls.

In the case of media failures, location databases at the leaf level must be reconstructed from scratch. Upon recovery, the database site can page all l-cells it covers to reconstruct its entries. Alternatively, entries may be reconstructed on demand, when a lookup arrives at the site. In addition, entries are added for objects in the l-cell as these objects place calls. Keeping backups can expedite the recovering procedure. However, forwarding pointers cannot be recovered as described. Instead, global searches are necessary to locate objects linked with forward pointers.

4. OTHER ISSUES

4.1. Replication

Many location strategies (e.g., [17, 18, 19]) involve replicating the location of a moving object in more than one site. Replication results in faster searches, since it increases the probability of finding a location entry at a neighbor site. However, faster searches are achieved at the expense of an increase in the cost of updates. When an object moves, all replicas maintaining its location become obsolete. Maintaining forwarding pointers reduces the overhead of updating replicas, since an object can still be located from its previous location following forwarding pointers. Selectively updating some of the replicas is also possible. That is, some of the replicas (e.g., the replicas at sites from which many calls originate) may be updated to point to the new location. Similarly, in the case of caching the callee's location at the caller's site (e.g., [11, 16]), if forwarding pointers are

used, cache entries remain valid even when the callee moves. Experiments (see Experiment 4 in Section 5) show that when replicas are kept at selective sites, the forwarding scheme outperforms the basic scheme even for high call to mobility ratios.

4.2. Advanced location queries

Besides location searches and updates, hierarchical location databases can be exploited for answering a wide range of advanced queries involving the location of moving objects. Such queries may be posed by mobile objects or their answer may involve mobile objects. An example is geographical multicasting [20] that involves sending messages to all objects currently in a specific geographical region. Other queries include queries for locating nearby objects (for instance mobile objects looking for specific services in their vicinity) or containment queries (for example, locating all objects in a specific region). Such queries can be efficiently answered using hierarchical databases, e.g. by querying databases at the region specified.

The introduction of forwarding pointers introduces a level of uncertainty in answering queries because location entries are not fully updated upon each move. Thus, answers to location queries are approximate. However, the uncertainty is bounded (or in the terms of [21] ignorance is bounded) depending on the condition for full updates. In particular, consider the first condition, according to which, location entries are fully updated to reflect the current location of an object, when the length of the chain of forwarding pointers exceeds some pre-specified limit k. Then the answer to a location query is guaranteed to be correct in the sense that the location returned as an answer was one of the k previous locations of the object. This can be seen as a time bound, the object was in the region in one of its k previous moves. The second condition of full updates depends on the distance traveled; location entries are fully updated when the lca of the current and previous location exceeds a specified threshold x. In this case the answer to the query is within a specified spatial limit in the sense that the object is currently in a site with a maximum distance depending on x from the location returned as an answer.

5. PERFORMANCE EVALUATION

We assume a hierarchy of location databases appropriately placed at the nodes of a given communication network. To allow for maximum flexibility in the design of the location management scheme, we consider hierarchies with a variable number of levels. The region covered by each leaf database corresponds to a unique physical address.

5.1. Cost estimation

We make a distinction between the database and the communication cost. We consider as database cost the total number of database operations (queries or updates). For the communication cost, we count the total number of links traversed by the messages involved. Assume that two nodes

u and v communicate with each other by traversing the network spanning tree connecting them. In general this spanning tree may be different from the hierarchical location database structure. Adjacent nodes in the hierarchical location database structure need to traverse the network interconnection spanning tree connecting them, which may contain several links.

Let $span(u, v)$ be the number of links in the path connecting u and v in this spanning tree. We assume that $span(u, v) = 2r\, lca(u, v)$, that is, it costs r times less, (i.e. there are r less links in the path) for nodes u and v to communicate directly than to communicate through their least common ancestor in the tree. Parameter r allows the modeling of different communication infrastructures, for example when the only way for two nodes to communicate is through the links of the tree, then $r = 1$. Let c be the l-cell of caller x, v be the current l-cell of object y, k be the length of the chain of forwarding pointers for object y, and n be the new l-cell inside which y moves. In the forwarding scheme, let v_0 be the l-cell containing the first forwarding pointer and v_i $(i = 0, \ldots, k)$ the i-th node in the chain.

We consider first a call placed by object x to object y. The corresponding costs are as follows.

- *Basic scheme:*
  database cost: $2\, lca(c, v) + 1$
  communication cost: $2\, lca(c, v) + span(c, v)$
- *Forwarding scheme:*
  database cost: $2\, lca(c, v_0) + k + 1$
  communication cost:

$$2\, lca(c, v_0) + \sum_{0}^{k-1} span(v_i, v_{i+1}) + span(c, v)$$

For a move of object y from the current l-cell v to a new l-cell n, the costs are:

- *Basic scheme:*
  database cost $2\, lca(v, n) + 1$
  communication cost $2\, lca(v, n) + 1 + span(v, n)$
- *Forwarding scheme:*
  database cost: 2
  communication cost: $span(v, n)$

There are additional costs for caching at calls or moves, fully updating the hierarchical scheme and deleting cycles, as follows:

- cost of caching at calls or moves:
  communication cost: $span(n, v_0)$
  database cost: 1
- cost of a full update of the hierarchical scheme resulting from a move outside the vicinity of the current region:
  communication cost:

$$2\, lca(n, v_0) + \sum_{0}^{k-2} span(v_i, v_{i+1})$$

database cost: $2\, lca(n, v_0) + k$

- cost of deleting a cycle (where l is the length of the cycle):

  communication cost:

$$\sum_{i=1}^{i=l} span(v_j, v_{j+1}) + span(n, v_0)$$

  database cost: l

5.2. Calling and mobility model

We assume that, for each object, calls and moves occur independently. The interarrival times between two calls follow an exponential distribution, with parameter the mean interarrival time between two calls, t_c. The interarrival times between two moves follow another exponential distribution, with parameter the mean interarrival time between two moves, t_m. The ratio of the number of calls over the number of moves called call to mobility ratio (*CMR*) is then

$$CMR = \frac{t_m}{t_c}.$$

The source of a call event is selected using one of the following distributions.

Arbitrary calls. A call may be placed from any l-cell with equal probability $1/n$, where n is the number of different l-cells. We use a discrete uniform distribution to select one from n l-cells.

Set of frequent callers. Each object receives most of its calls from a specific set of locations. This corresponds to a real-life situation in which an object is frequently called by a set of other objects or groups of objects (e.g. in the case of mobile users, friends, family, business associates or regular customers, or in the case of mobile software, associated code). We model a set of frequent callers with a discrete bimodal distribution, which distributes a P_f probability uniformly over a set of specific locations and a $1 - P_f$ probability uniformly over all other locations. So, a call has P_f/n_f probability to be placed by a specific frequent calling location, and $(1 - P_f)/(n - n_f)$ to be placed by another location, where n_f is the number of frequent calling locations.

The destination of a move event is selected via one of the following distributions.

Arbitrary moves. An object may move to any location, except from its current one, with the same probability. We use a uniform distribution as in the case of arbitrary calls; however, the probability that the object remains in the same location after a move is 0.

Frequent moves to neighbor cells on a grid. Since objects usually move to nearby locations, we model such a situation in which distant moves are unlikely to happen and short moves to neighbor locations are most likely to happen. We assume that the object moves on a grid of l-cells (see Figure 7) and that the object may move with a high probability (P_N) to a neighbor l-cell on the grid while there
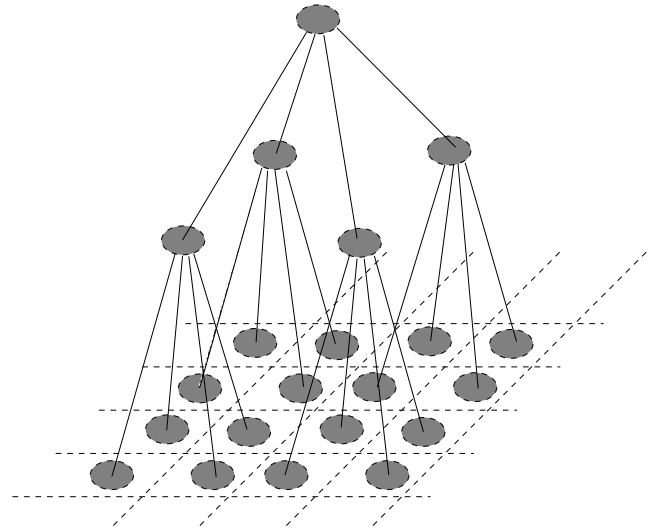
FIGURE 7. Movement to adjacent cells on a grid.

is a small probability $(1 - P_N)$ that the object will jump to some other non-neighbor l-cell. In case of wireless mobile computing, the movement to neighbor l-cells corresponds to an active user physically moving across l-cells. On the other hand, the rare movement to non-neighbor l-cells, corresponds to a user turning off its mobile host and turning it on again in some arbitrary l-cell. To model this distribution, we use a discrete bimodal probability distribution similar to that used for the set of frequent callers.

5.3. Experiments and results

We have developed an event-driven simulator to evaluate the performance of the location strategies. We simulate calls to a specific mobile object and moves made by this object using an event-driven simulator. An event is either a move or a call event. The simulator software has been developed in C++ and runs on a Sun Sparc 10 workstation. The purpose of the set of experiments is to

- determine the optimal condition for initiating full updates,
- demonstrate the benefits of caching,
- provide a comparison of the forwarding scheme with the basic scheme, and
- illustrate optimizations of the forwarding scheme for frequent callers.

We run the experiments for a wide range of call to mobility ratios and for a total of about 6000 move and call events per object.

More specifically, results for Experiments 1–3 are presented for small *CMR*s 0.01–1.0 (highly mobile users), since for larger *CMR*s the benefits of forward pointers are diminished. Experiment 4 shows that when the call model is a set of frequent callers, the benefits of our scheme are maintained even for large *CMR*s (0.01–100).

We have experimented with hierarchies of different height and width. The results show that the relative performance

TABLE 2. Parameters of the experiments.

Parameter	Description	Value
CMR	Call to mobility ratio for the mobile user	Varies
Database cost	Total number of database (query and update) operations	Output
Communication cost	Weighted sum of tree and forwarding pointer traversals (weighted by distance measured by the level of the lca of the two locations)	Output
n_f	Number of frequent caller locations	15
P_f	Percentage of calls from frequent callers (set of frequent callers model)	80%
P_n	Probability of moving to an adjacent location (frequent moves to neighbor cells model)	95%
r	Models the relative cost of following a forwarding pointer vs. following a tree link	1/3
Topology Parameters		
Height		8
Node outdegree		4
n	Number of leaf nodes	~65,000

of the schemes under consideration remains the same. The results presented are for a tree of height 8 and out-degree 4 [22]. The experiments were performed multiple times and statistical means were derived for all estimates. Since the relative size of the communication and the database cost depends on many factors such as the load of the network and the size of the databases, we consider the two costs separately. The value of r affects the communication cost of the forwarding scheme which increases with r. For the experiments, we set the communication factor r equal to 1/3. This is a realistic assumption, because in practice, even when traversing a forwarding pointer between two nodes means following the tree path through their LCA, it costs less, since there is no need for establishing connections and querying at each tree node on the path. Table 2 summarizes the parameters for the experiments.

Experiment 1: condition for full updates

The first experiment aims at determining an appropriate condition for initiating full updates. We considered both conditions set in Section 2, i.e. (a) the condition based on the distance of the move, as determined by the level (height) of the least common ancestor of the new and old location of the move operation, and (b) the condition based on the length of the chain of forwarding pointers. In particular, we are looking for an optimal value for the level and length beyond which to fully update so that the overall database and communication cost is minimized. Table 3 shows the optimal full update conditions for nearby moves, and Table 4 for uniform moves. The optimal values of length and height were obtained by performing the same experiment for different values of the height and length and selecting the ones for which the communication/database cost was the minimum.

The optimal purge conditions are different depending on

whether caching is used or not. With caching, full updates need to be performed less frequently than without caching, since the cost of calls is reduced by using the cached value.

As expected, the optimal values directly depend on the *CMR*, since fully updating increases the cost of moves, while it decreases the cost of calls. Also, the optimal values are slightly higher for uniform moves. This is because basic moves cost more in the uniform model of moves than in the nearby model of moves and thus the savings from forwarding are correspondingly higher.

For uniform moves the optimal purge condition for height (8) coincides with the overall height of the tree, meaning that a full update has to be performed even less frequently than given by this condition. This is expected since when the move distribution is uniform, the probability of a move traversing the root of the tree is $\frac{1}{2}$, meaning that full update will occur every second move on average. Thus, the full update based on the height condition is not appropriate for uniform moves.

Experiment 2: benefits of caching

We have performed a set of experiments to verify the benefits of caching. For this set of experiments, the condition for fully updating was based on the optimal values obtained at Experiment 1. Caching adds no overhead to the latency of move or call operations, since it can be performed off-line. Figure 8 demonstrates the effectiveness of caching for nearby moves and Figure 9 for uniform moves. Caching works better in the case of nearby moves, since there is locality. The mean and maximum chain lengths refer to the mean and maximum number of pointers a call has to follow before locating the callee. As shown, caching proves very effective in minimizing both the mean and the maximum values of the chain. Besides the call set-up time, we have also computed the overall database and communication cost

TABLE 3. Optimal values for purging in terms of the overall database and communication cost for nearby moves (left) with caching and (right) without caching.

CMR	Optimal height		Optimal length			CMR	Optimal height		Optimal length	
	Database cost	Comm cost	Database cost	Comm cost			Database cost	Comm cost	Database cost	Comm cost
0.01	8	8	20	20		0.01	8	8	20	20
0.05	8	8	13	12		0.05	8	7	13	12
0.1	8	7	11	8		0.1	7	6	10	8
0.2	8	7	8	5		0.2	5	4	7	5
0.3	8	7	5	4		0.3	4	4	5	4
0.4	8	6	5	4		0.4	4	3	5	3
0.5	8	6	4	3		0.5	4	3	4	2
0.6	8	4	4	2		0.6	3	3	3	2
0.8	8	4	4	2		0.8	3	3	3	2
1.0	8	4	3	1		1.0	3	3	2	1

(cache)   (no cache)

TABLE 4. Optimal values for purging in terms of the overall database and communication cost for uniform moves (left) with caching and (right) without caching.

CMR	Optimal height		Optimal length			CMR	Optimal height		Optimal length	
	Database cost	Comm cost	Database cost	Comm cost			Database cost	Comm cost	Database cost	Comm cost
0.01	8	8	20	20		0.01	8	8	20	20
0.05	8	8	20	11		0.05	8	8	20	10
0.1	8	8	20	9		0.1	8	8	14	6
0.2	8	8	19	6		0.2	8	8	13	5
0.3	8	8	19	5		0.3	8	8	9	4
0.4	8	8	19	4		0.4	8	8	8	3
0.5	8	8	18	3		0.5	8	8	7	3
0.6	8	8	18	3		0.6	8	8	6	2
0.8	8	8	17	3		0.8	8	8	5	2
1.0	8	8	17	2		1.0	8	8	4	2

(cache)   (no cache)

FIGURE 8. Effectiveness of caching for nearby moves.
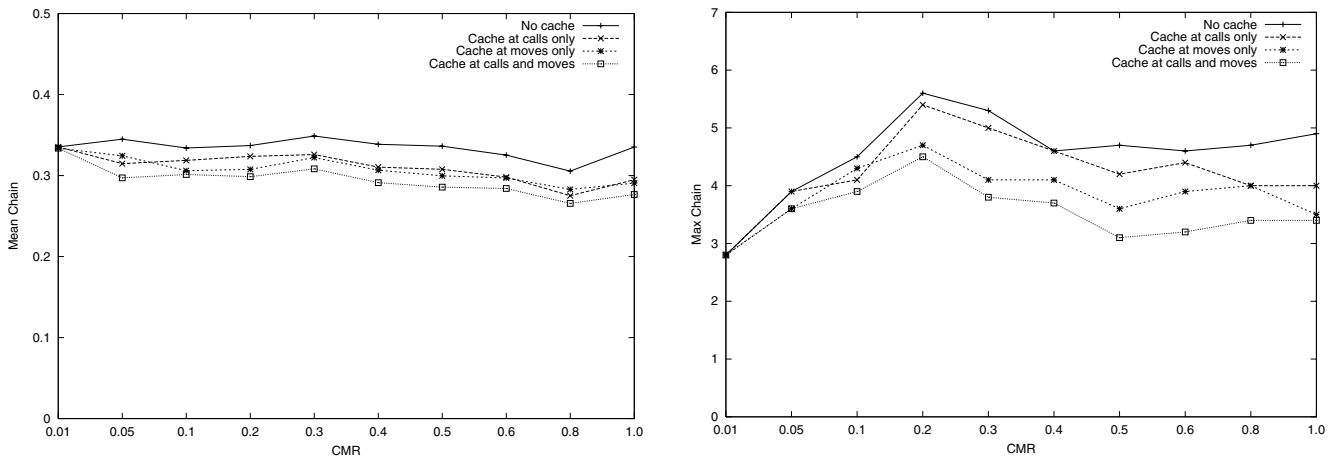
FIGURE 9. Effectiveness of caching for uniform moves.

(Figure 10 for nearby moves and Figure 11 for uniform moves). As shown, caching does not add to the overall database and communication cost.

In particular, for uniform moves, the overall database and communication cost are more or less the same for all schemes, since the benefits of caching in the call set-up time is balanced with the increase in the overall time.

Experiment 3: forward versus basic

We have performed a number of experiments to study the effect of the forwarding scheme in decreasing the overall database load and communication traffic. Depending on the *CMR* both the overall database traffic and the communication cost are decreased by a factor of 52% to a factor of 5% for nearby moves (Figure 12(left)). For uniform calls, the corresponding values ranges from around 18% to around 7% (Figure 12(right)).

The price for this improvement is an increase of the call set up time. This is shown in Figure 13(left) for nearby moves and in Figure 13(right) for uniform moves. Call latency can be effectively kept short if information about the calling behavior is known beforehand. In this case, a simple scheme that caches the location of the object at the callers can reduce the cost. Next, we present results for such a scheme.

Experiment 4: set of frequent callers

For the purposes of this experiment, we assume a pre-specified set of frequent callers, uniformly chosen among all l-cells. In practice, either each mobile object explicitly specifies this set, or the set is derived by observing for each mobile object the frequency of receiving calls. We study how we can benefit from the peculiarity of this model and reduce the overall cost and the call set-up time by extending both the basic and the forwarding schemes with lazy replication.

In particular, the location of a mobile object is replicated at the l-cell of its frequent callers. To locate an object, a frequent caller first uses the location saved at its region. If

this location is outdated (e.g. the object has in the meantime moved to a new location) a location miss is signaled. Upon a miss, the caller uses the usual procedure to locate the callee. An outdated replica of an object is updated upon a location miss, after the current location of the object is found by using the normal procedure. Setting forwarding pointers improves the performance of replication since it decreases the location miss ratio. Figure 14 shows the benefits of forwarding in terms of the overall cost, while Figure 15 shows the benefit of forwarding in terms of the call set-up time.

Note, that normally forwarding is tuned for highly mobile users, thus it does not perform well for high *CMR* values. However, in the case of frequent callers, the forwarding scheme works well even for high values of the *CMR*, since the benefits of the lazy evaluation balance the cost increase of the forwarding schemes.

6. RELATED WORK

Location management has attracted much current research in mobile computing. Various approaches have been proposed for reducing the cost of move and call operations for both hierarchical and non-hierarchical location database architectures [23, 3]. See for example [24] for a survey.

One proposal to reduce the cost of calls is based on replicating (see e.g. [18, 19] for hierarchical and [17] for non-hierarchical architectures) or caching (see e.g. [16] for non-hierarchical and [11] for hierarchical architectures) the location of mobile users at the sites of their frequent callers. To reduce the cost of network traffic generated by frequent updates, a scheme based on partitioning is presented in [25, 26]. Partitioning is performed by grouping the cells among which a mobile unit moves frequently and separating the cells between which it relocates infrequently. Only moves across partitions are reported. A hierarchical method for location binding in wide-area systems is used in the Globe wide-area location service [12, 15]. Globe advocates a combination of caching and partitioning. An interesting feature is that an object, being a service, may have more than one location. Finally, the assignment of location databases
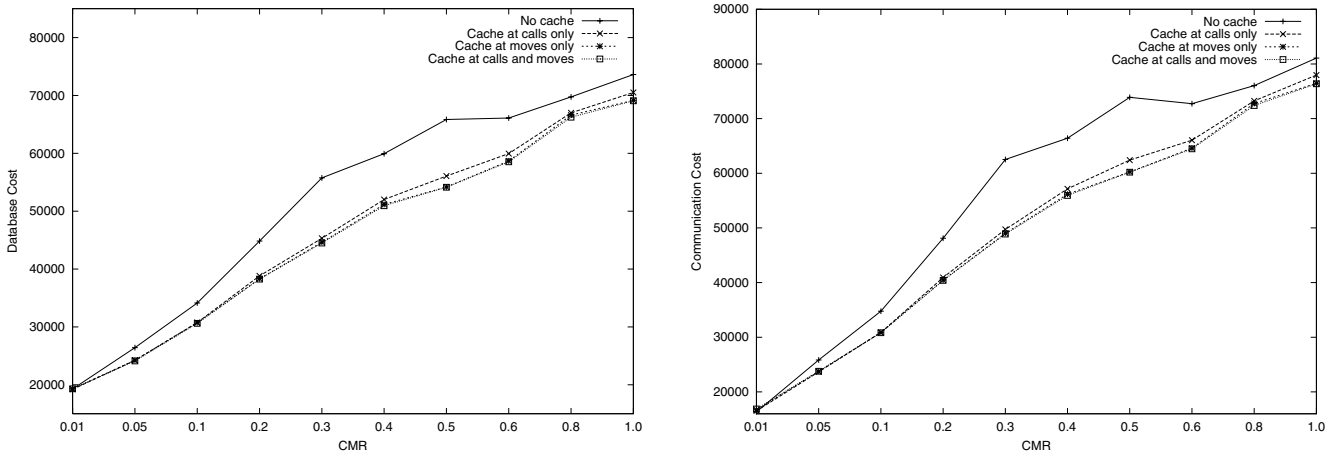
FIGURE 10. Overall database and communication cost for nearby moves.
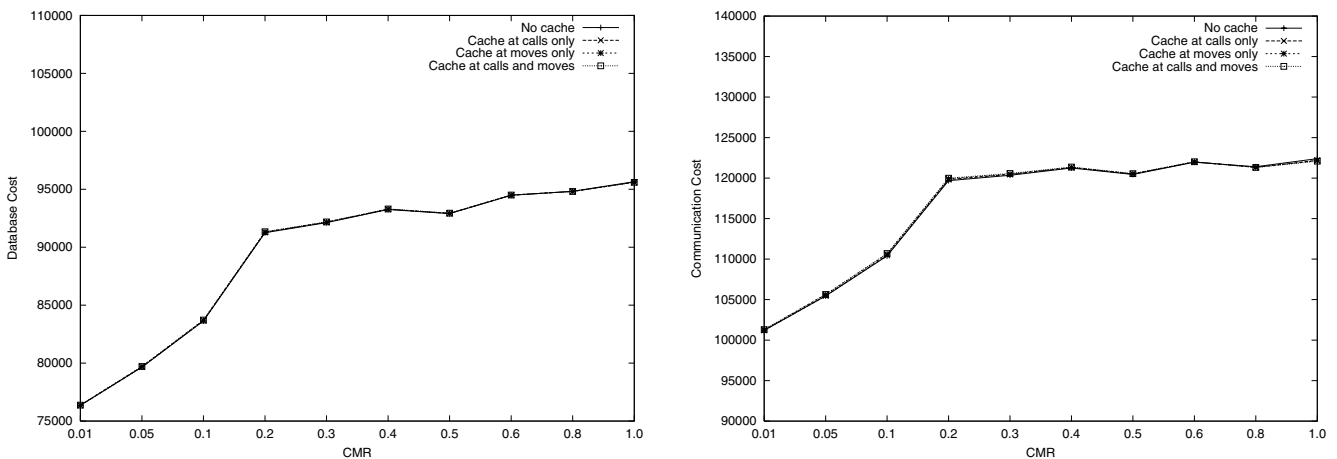

FIGURE 11. Overall database and communication cost for uniform moves.

to the nodes of a signaling network is discussed in [10], where database placement is formulated as an optimization problem. Those schemes are orthogonal to the address forwarding strategy and can be used in conjunction with it to further reduce database and communication costs.

Address forwarding for a non-hierarchical architecture is discussed in [27] along with an analysis of its benefits for different *CMR*s. Forwarding pointers have been also used in [28] again for non-hierarchical location databases. The focus there is on IP-based protocols. In [29], forwarding is explored in a different framework for a hierarchical, though not tree-structured, location database architecture, called regional matching directory. The treatment is theoretical and based on a worst case order analysis.

A forwarding pointer technique for a two-tier architecture is also proposed in the local anchoring scheme [30], where a forwarding pointer chain of length at most one is maintained. In particular, for each mobile object, a VLR close to the object is selected as its local anchor (LA). In some cases, the LA may be the same as the mobile object's serving VLR. Otherwise, the LA maintains a forwarding pointer to the object's current VLR. For each mobile object, the HLR maintains its serving LA. To locate a mobile unit, the HLR

is queried first and then the associated LA is contacted. If the LA happens to be the serving VLR, no further querying is necessary, else the forwarding pointer at the LA is used to locate the mobile unit. Since after a call delivery, the HLR knows the current location of a mobile unit, the HLR is always updated after a call to record the current VLR. Depending on whether the HLR is updated upon a move, two schemes are proposed: static and dynamic local anchoring. In static local anchoring, the HLR is never updated at a move. In dynamic local anchoring, the serving VLR becomes the new LA if this will result in lower expected costs.

A non-hierarchical forwarding scheme, SSP chains [31], has been proposed for transparently migrating object references between processes in distributed computing. The SSP-chain short-cutting technique is somewhat similar to our caching at calls method.

In [32], a form of forwarding pointers is used in a hierarchical location architecture. Their hierarchical architecture is different from ours, in that instead of storing at each internal database a pointer to the corresponding lower level database, the actual location is stored. That is, in effect, the location of each mobile object is replicated at all nodes on the path from the root to the leaf node covering
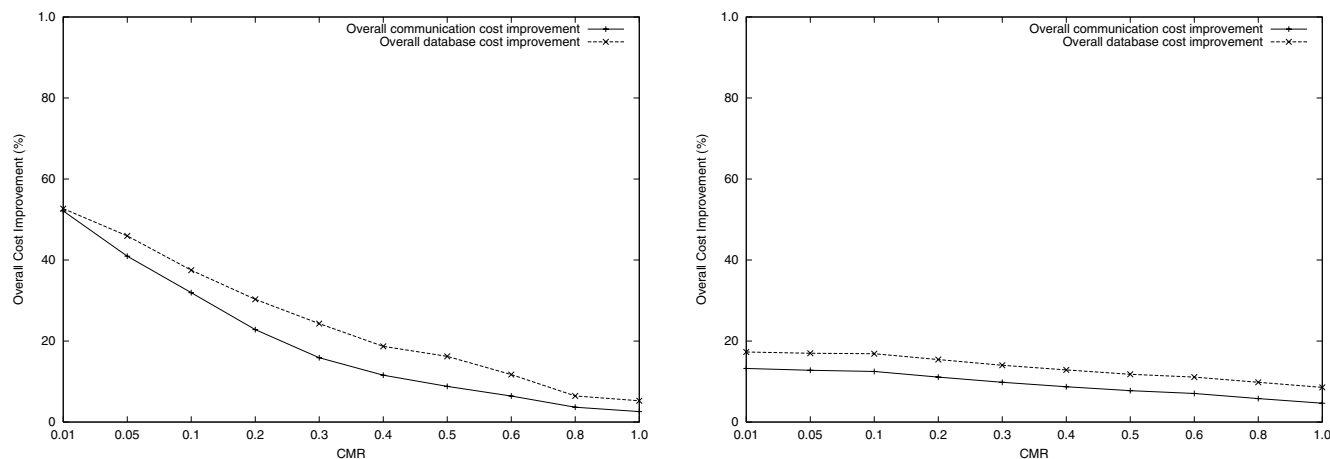
FIGURE 12. Decrease of the database and communication traffic caused by forwarding (left) for nearby moves and (right) for uniform moves.
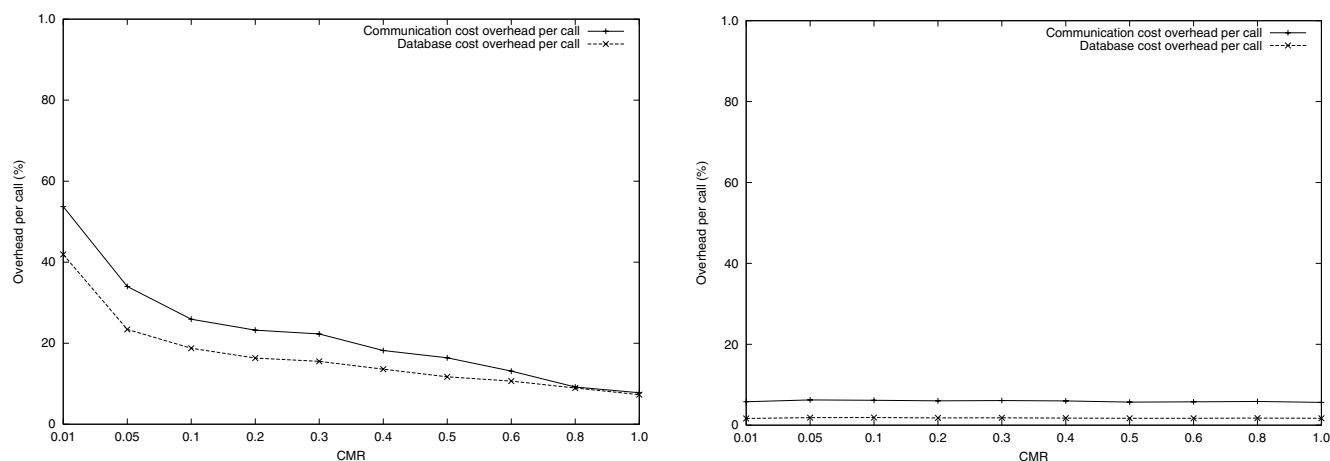

FIGURE 13. Increase of the call set-up time (left) for nearby moves and (right) for uniform moves.

the object's current location. Furthermore, the emphasis of our work is on developing techniques for enhancing the basic pointer forwarding scheme, while the emphasis in [32] is on designing an adaptive scheme that will dynamically decide which location scheme to deploy based on the past behavior of the system. Our scheme can be extended so that forwarding pointers can be selectively used based on the call to mobility ratio and other system parameters.

Databases that record the location of moving objects are also discussed in [33, 34]. However, in this work a single centralized database is considered. The focus is on determining when to update the location of a moving object. The approach advances an information cost model that captures uncertainty, deviation and communication.

In terms of concurrency control, there has been a lot of research on databases, including concurrency control for tree-based indexes. Such concurrency control schemes exploit the specific characteristic of each index structure to achieve more concurrency. For example, in B+-trees, updates occur only when a node becomes full and it must be split into two sibling nodes. In this case, a B-link tree [35, 36] (a variant of a B+-tree that uses links to chain

all nodes at each level together) exhibits more concurrency, since a look-up that arrives at a newly-split node can locate an entry by just following the links to a sibling node. The concurrency control schemes in this paper are tailored to the proposed hierarchical structure.

A locking protocol for location queries is presented in [37]. Their hierarchical structure is different from ours. Location information is stored only in leaf nodes; internal nodes just collect location information from queries that they submit to lower level nodes. While we consider the concurrent execution of updates and look-ups, they consider concurrent updates and aggregate queries of the form: count the number of mobile users currently in the area covered by some node x. Before processing an aggregate query of this type, locks must be requested and acquired on all nodes in the subtree rooted at x. An update is performed only if the associated leaf nodes are not locked. Otherwise, the update is deferred and a message is propagated to the root making it responsible for updating the leaf nodes. To avoid the overhead of propagating the message to the root, it is also suggested to just mark the entries to be updated as obsolete and let the aggregate query use this information accordingly.
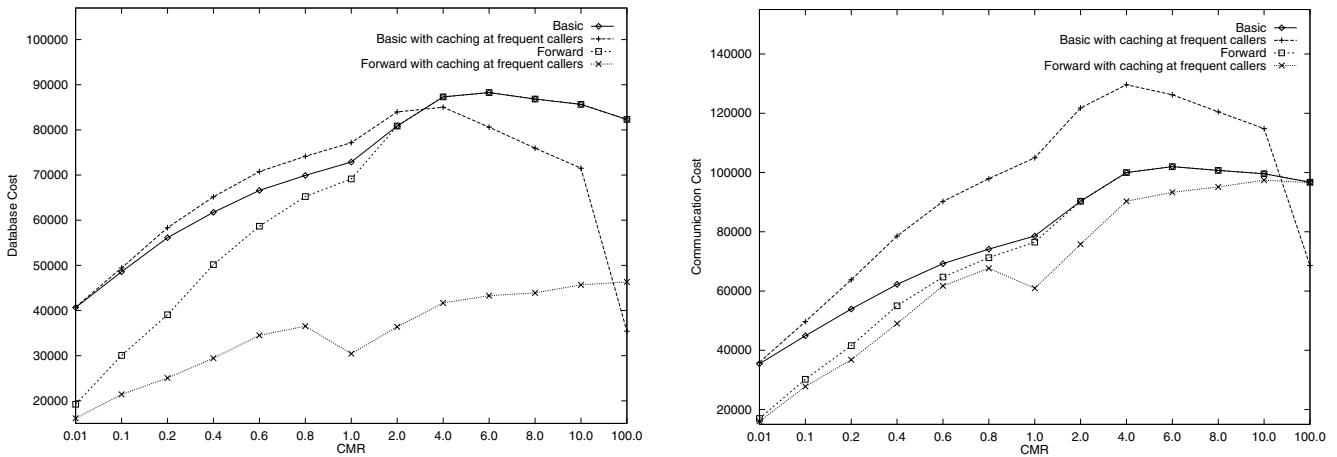
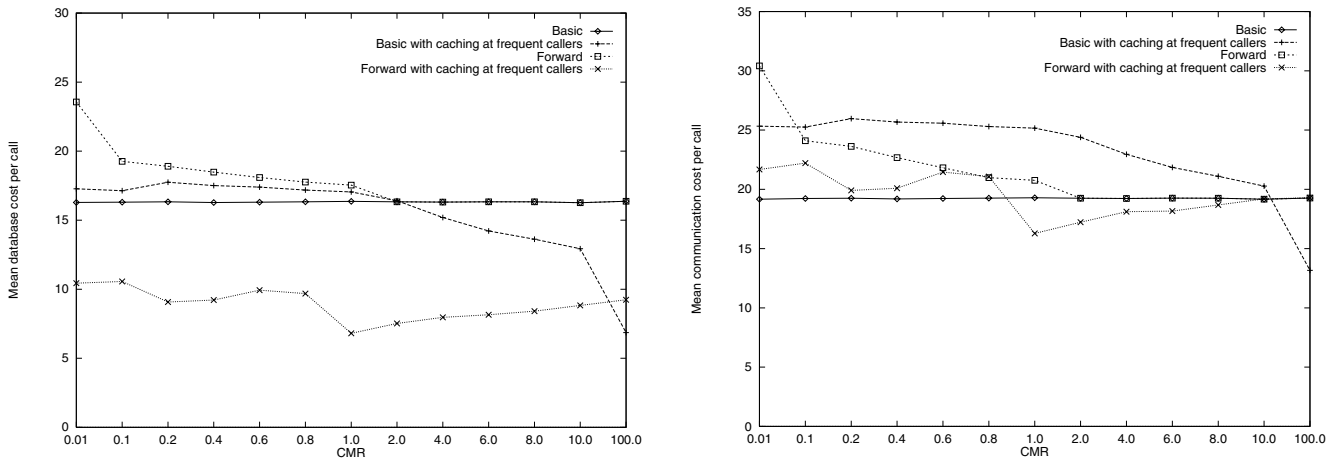**FIGURE 14.** Overall database and communication cost for the set of frequent callers.

FIGURE 15. Call set-up time for the set of frequent callers.

7.   SUMMARY

Tracking objects is central to mobile computing. In this paper, we have studied the use of forwarding pointers in a hierarchical database arrangement. In such schemes, instead of fully updating the hierarchical database upon each location update, a forwarding pointer to the new location is set at the previous location of the object. However, the forwarding scheme increases the cost of location searches, since a chain of forwarding pointers may have to be traversed before locating an object. To effectively reduce the length of such chains, we have introduced caching techniques as well as conditions for initiating an update of all associated database entries. Finally, we have described a synchronization method to control the concurrent execution of location searches and updates and a protocol to handle site failures. Experimental results show that under certain assumptions and for small call to mobility ratio, the proposed forwarding scheme leads in reduction of both the overall database load and the communication traffic by 20% to 60% depending on the call to mobility ratio and the moving behavior.

A preliminary version of our work has appeared in [38].

REFERENCES

[1] Forman, G. H. and Zahorjan, J. (1994) The challenges of mobile computing. *IEEE Comput.*, **27**, 38–47.

[2] Imielinski, T. and Badrinath, B. R. (1994) Wireless mobile computing: challenges in data management. *Commun. ACM*, **37**, 18–28.

[3] Pitoura, E. and Samaras, G. (1998) *Data Management for Mobile Computing*. Kluwer Academic, Dordrecht.

[4] Riecken, D. (ed.) (1994) Special issue on intelligent agents. *Commun. ACM*, **37**(7).

[5] Vitek, J. and Tschudin, C. (eds) (1997) *Mobile Object Systems: Towards the Programmable Internet. Lectures Notes in Computer Science*, **1222**. Springer, Heidelberg.

[6] Huhns, M. N. and Singh, M. P. (eds) (1997) Special issue on internet-based agents. *IEEE Internet Comput.*, **1**(4).

[7] Weiser, M. (1993) Some computer science issues in ubiquitous computing. *Commun. ACM*, **36**, 75–84.

[8] Baentsch, M., Baum, L., Molter, G., Rothkugel, S. and Sturm, P. (1997) Enhancing the web's infrastructure: from caching to replication. *IEEE Internet Comput.*, **1**, 18–27.

[9] Wang, J. Z. (1993) A fully distributed location registration strategy for universal personal communication systems. *IEEE J. Selected Areas Commun.*, **11**, 850–860.

[10] Anantharam, V., Honig, M. L., Madhow, U. and Kei, V. K. (1994) Optimization of a database hierarchy for mobility tracking in a personal communications network. *Performance Evaluation*, **20**, 287–300.

[11] Jain, R. (1996) Reducing traffic impacts of PCS using hierarchical user location databases. In *Proc. IEEE Int. Conf. on Communications*, June, Dallas, TX. IEEE Computer Society Press, Los Alamitos, CA.

[12] van Steen, M., Hauck, F. J., Homburg, P. and Tanenbaum, A. S. (1998) Locating objects in wide-area systems. *IEEE Commun. Mag.*, **1**, 2–7.

[13] Mouly, M. and Pautet, M. B. (1992) The GSM System for Mobile Communications. *Mouly Consultant*, June.

[14] Eynard, C., Lenti, M., Lombardo, A., Marengo, O. and Palazzo, S. (1995) A methodology for the performance evaluation of data query strategies in Universal Mobile Telecommunication Systems (UMTS). *IEEE J. Selected Areas Commun.*, **13**, 839–907.

[15] van Steen, M., Hauck, F. J., Ballintijin, G. and Tanenbaum, A. S. (1998) Algorithmic design of the globe wide-area location service. *Comp. J.*, **41**, 297–310.

[16] Jain, R., Lin, Y-B., Lo, C. and Mohan, S. (1994) A caching strategy to reduce network impacts of PCS. *IEEE J. Selected Areas Commun.*, **12**, 1434–1444.

[17] Jannink, J., Lam, D., Shivakumar, N., Widom, J. and Cox, D. C. (1996) Efficient and flexible location management techniques for wireless communication systems. In *Proc 2nd ACM Int. Conf. on Mobile Computing and Networking (Mobicom'96)*, pp. 38–49. ACM Press, New York.

[18] Rajagopalan, S. and Badrinath, B. R. (1995) An adaptive location management strategy for mobile IP. In *Proc. 1st ACM Int. Conf. on Mobile Computing and Networking (Mobicom'95)*, Berkeley, CA. ACM Press, New York.

[19] Shivakumar, N. and Widom, J. (1995) User profile replication for faster location lookup in mobile environments. In *Proc. 1st ACM Int. Conf. on Mobile Computing and Networking (Mobicom'95)*, pp. 161–169. ACM Press, New York.

[20] Navas, J. C. and Imielinski, T. (1997) Geographic addressing and routing. In *Proc. 3rd ACM/IEEE Int. Conf. on Mobile Computing and Networking (MobiCom'97)*, Budapest, Hungary, September 1997, pp. 26–30. ACM Press, New York.

[21] Imielinski, T. and Badrinath, B. R. (1992) Querying in highly mobile distributed environments. In *Proc. 18th Int. Conf. on Very Large Data Bases (VLDB 92)*, pp. 41–52. Morgan Kaufmann, San Francisco, CA.

[22] Stanford Pleiades Research Group. Stanford University Mobile Activity TRAces (SUMATRA). Available at: www-db.stanford.edu/sumatra.

[23] Mohan, S. and Jain, R. (1994) Two user location strategies for personal communication services. *IEEE Personal Commun.*, **1**, 42–50.

[24] Pitoura, E. and Samaras, G. (1998) *Locating Objects in Mobile Computing*. Technical Report TR: 98-20, University of Ioannina, Computer Science Dept. *IEEE TKDE*, accepted. Also available at: www.cs.uoi.gr/˜pitoura/pub.html.

[25] Badrinath, B. R., Imielinski, T. and Virmani, A. (1992) Locating strategies for personal communications networks. In *Proc. 1992 Int. Conf. on Networks for Personal Communications*. IEEE Globecomm, FL.

[26] Cho, G. and Marshall, L. F. (1995) An efficient location and routing schema for mobile computing environments. *IEEE J. Selected Areas Commun.*, **13**, 868–879.

[27] Jain, R. and Lin, Y.-B. (1995) An auxiliary user location strategy employing forwarding pointers to reduce network impacts of PCS. *Wireless Networks*, **1**, 197–210.

[28] Ioannidis, J., Duchamp, D. and Maguire Jr, G. Q. (1991) IP-based protocols for mobile internetworking. In *Proc. ACM SIGCOMM Symposium on Communications, Architectures and Protocols*, pp. 235–245. ACM Press, New York.

[29] Awerbuch, B. and Peleg, D. (1991) Concurrent online tracking of mobile users. In *Proc. SIGCOMM 91*, pp. 221–233. ACM Press, New York.

[30] Ho, J. S. M. and Akyildiz, I. F. (1996) Local anchor scheme for reducing signalling cost in personal communication networks. *IEEE/ACM Trans. Networking*, **4**, 709–725.

[31] Shapiro, M., Dickman, P. and Plainfosse, D. (1992) *SSP Chains: Robust, Distributed References Supporting Acyclic Garbage Collection*. Technical Report 1799, INRIA, Rocquentcourt, France.

[32] Krishna, P., Vaidya, N. H. and Pradhan, D. K. (1996) Static and dynamic location management in mobile wireless networks. *J. Comp. Commun. (special issue on Mobile Computing)*, **19**, 321–334.

[33] Sistla, A. P., Wolfson, O., Chamberlain, S. and Dao, S. (1997) Modeling and querying moving objects. In *Proc. 13th Int. Conf. on Data Engineering (ICDE 97)*, pp. 422–432. IEEE Computer Society Press, Los Alamitos, CA.

[34] Wolfson, O., Chamberlain, S., Dao, S., Jiang, L. and Mendez, G. (1998) Cost and imprecision in modeling the position of moving objects. In *Proc. 14th Int. Conf. on Data Engineering (ICDE 98)*, pp. 588–596. IEEE Computer Society Press, Los Alamitos, CA.

[35] Lehman, P. and Yao, S. (1981) Efficient locking for concurrent operations on B-trees. *ACM Trans. Database Syst.*, **6**, 650–670.

[36] Sagiv, Y. (1985) Concurrent operations on B*-trees with overtaking. In *Proc. 4th ACM Symp. on Principles of Database Systems*, pp. 28–37. ACM Press, New York.

[37] Budiarto, Harumoto, K., Tsukamoto, M. and Nishio, S. (1997) Position locking: handling location dependent queries in mobile computing environment. In *Proc. WWCA97. Lectures Notes in Computer Science*, **1274**, 363–378.

[38] Pitoura, E. and Fudos, I. (1998) An efficient hierarchical scheme for locating highly mobile users. In *Proc. 7th Int. Conf. on Information and Knowledge Management (CIKM'98)*, pp. 218–225. ACM Press, New York.